



AFRL-RI-RS-TR-2017-035

## RESILIENT DIFFUSIVE CLOUDS

---

TRUSTEES OF DARTMOUTH COLLEGE

*FEBRUARY 2017*

FINAL TECHNICAL REPORT

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED*

STINFO COPY

**AIR FORCE RESEARCH LABORATORY  
INFORMATION DIRECTORATE**

## **NOTICE AND SIGNATURE PAGE**

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2017-035 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

**/ S /**

ANNA L. WEEKS  
Work Unit Manager

**/ S /**

WARREN H. DEBANY, JR.  
Technical Advisor, Information  
Exploitation and Operations Division  
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

<b>REPORT DOCUMENTATION PAGE</b>				<b>Form Approved OMB No. 0704-0188</b>	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. <b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b>					
<b>1. REPORT DATE (DD-MM-YYYY)</b> FEB 2017		<b>2. REPORT TYPE</b> FINAL TECHNICAL REPORT		<b>3. DATES COVERED (From - To)</b> SEP 2011 – SEP 2016	
<b>4. TITLE AND SUBTITLE</b>  RESILIENT DIFFUSIVE CLOUDS				<b>5a. CONTRACT NUMBER</b> FA8750-11-2-0257	
				<b>5b. GRANT NUMBER</b> N/A	
				<b>5c. PROGRAM ELEMENT NUMBER</b> 62722F	
<b>6. AUTHOR(S)</b>  Stephen Taylor				<b>5d. PROJECT NUMBER</b> MRC0	
				<b>5e. TASK NUMBER</b> DA	
				<b>5f. WORK UNIT NUMBER</b> RT	
<b>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</b> Trustees of Dartmouth College 11 Rope Ferry Road, #6210 Hanover, NH 03755-1404				<b>8. PERFORMING ORGANIZATION REPORT NUMBER</b>	
<b>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</b>  Air Force Research Laboratory/RIGB 525 Brooks Road Rome NY 13441-4505				<b>10. SPONSOR/MONITOR'S ACRONYM(S)</b>  AFRL/RI	
				<b>11. SPONSOR/MONITOR'S REPORT NUMBER</b>  AFRL-RI-RS-TR-2017-035	
<b>12. DISTRIBUTION AVAILABILITY STATEMENT</b> Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.					
<b>13. SUPPLEMENTARY NOTES</b>					
<b>14. ABSTRACT</b>  The primary goal of this research project was to explore an alternative to conventional operating systems based on <i>diffusive management of diversified virtual machines</i> . The concepts lead to a view of cloud computing in which vulnerabilities are different at every host, attackers cannot perform reliable surveillance and attacks are unable to persist on the timescale of military missions. These concepts stand in stark contrast to today's systems where vulnerabilities are amplified by being present at every host in the cloud, systems are static allowing long-term surveillance, and operating systems are pre-determined and static allowing kernel implants to persist.					
<b>15. SUBJECT TERMS</b> Software Based Biometrics; Cognitive Fingerprint; Continuous User Identification					
<b>16. SECURITY CLASSIFICATION OF:</b>			<b>17. LIMITATION OF ABSTRACT</b>  UU	<b>18. NUMBER OF PAGES</b>  36	<b>19a. NAME OF RESPONSIBLE PERSON</b> ANNA L. WEEKS
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			<b>19b. TELEPHONE NUMBER (Include area code)</b> N/A

## TABLE OF CONTENTS

Section	Page
List of Figures.....	ii
List of Tables.....	iii
ACKNOWLEDGEMENTS .....	iv
1.0 SUMMARY.....	1
2.0 INTRODUCTION.....	1
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES .....	3
3.1 ASSUMPTIONS .....	3
3.2 METHODS AND PROCEDURES .....	5
3.2.1 Core Ideas .....	5
3.2.2 Utility Virtual Machines.....	6
3.2.3 Diffusive Scheduling .....	9
3.2.4 VT-d.....	14
3.2.5 ExOShim .....	14
3.2.6 KPLT .....	14
3.2.7 Load-Time Diversity.....	15
3.2.8 Compile-Time Diversity.....	16
3.2.9 Replication Diversity.....	17
3.2.10 Diversified-NFS.....	17
3.2.11 Asymmetric Multiprocessing.....	19
4.0 RESULTS AND DISCUSSION .....	20
4.1 Core Results .....	21
4.2 Diffusive Scheduling .....	22
5.0 CONCLUSIONS .....	25
6.0 REFERENCES .....	26
7.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS .....	29

## LIST OF FIGURES

Figure		Page
1	APT Thread Model .....	3
2	Timeliness in the Attack Process .....	4
3	Bare-Metal Hypervisor .....	6
4	System based on Utility Virtual Machines .....	7
5	Scheduling Software Components .....	10
6	Code to Schedule Next Process .....	12
7	Dynamic Load-Balancing Code .....	13
8	Original Function with two vacuous padded variants .....	17
9	D-NFS System Overview .....	18
10	Diffusion Performance as Interrupt Heat Rises .....	24

## LIST OF TABLES


Table		Page
1	Memory and Processor Benchmarks .....	21
2	Scheduler Performance Characteristics .....	23

## ACKNOWLEDGEMENTS

The work described in this report represents the collective work of a talented group of graduate students at Dartmouth College. Within the strategic framework, much of the detail is taken directly from the writings of these students. In any group effort there is naturally considerable collaboration and overlap, the following gives a rough breakdown of the primary activities each student was engaged in; in alphabetic order: Scott Brookes (signals, diversity, memory systems, asymmetric multiprocessing, ExOShim, KPLT), Jason Dahlstrom (hardware-hiding), Robert Denz (kernel, hypervisor, UVM's, diffusive scheduling, ExOShim, KPLT, D-NFS), Michael Henson (memory encryption), Morgon Kanter (hypervisor, camouflage, diversity), Stephen Kuhn (hypervisor, networking, VT-d, network hiding, forensics), Kathleen McGill (rMP and resilience), Colin Nichols (kernel and hypervisor), Martin Osterloh (systems engineering, reverse engineering, D-NFS, KPLT, ExOShim).

## DATA RIGHTS

As per contract FA8750-11-2-0257, pertaining to the research described in this report, Dartmouth College grants to the U.S. Government a royalty free, world-wide, non-exclusive, irrevocable license to use, modify, reproduce, release, perform, display, or disclose any data for Government purposes.

  
9/12/2016.

## 1.0 SUMMARY

The primary goal of this research project was to explore an alternative to conventional operating systems based on *diffusive management of diversified virtual machines*. The concepts lead to a view of cloud computing in which vulnerabilities are different at every host, attackers cannot perform reliable surveillance and attacks are unable to persist on the timescale of military missions. These concepts stand in stark contrast to today's systems where vulnerabilities are amplified by being present at every host in the cloud, systems are static allowing long-term surveillance, and operating systems are pre-determined and static allowing kernel implants to persist.

The research builds upon and complements operating system research recently concluded under the DARPA CRASH program as part of the "Attacking Time" effort. It leverages small-footprint hypervisor and micro-kernel designs that incorporate non-deterministic methods and techniques to *increase attacker workload* and *operate through attacks* even if the attacks are never detected.

Collectively, these two DARPA projects have resulted in a new way to structure distributed systems based on a *non-deterministic defense-in-depth*. This defense combines a series of breakthrough technologies that collectively provide an insurmountable barrier to the tactical viability of Advanced Persistent Threats (APT's).

## 2.0 INTRODUCTION

Our method for mitigating APT's is based on a *non-deterministic defense-in-depth* in which a collection of innovative technologies are applied, either in isolation or in combination, to successively increase attacker workload and operate through attacks. These techniques prevent an adversary from operating on timescales that lie within the tempo of US military operations.

Whereas the CRASH effort [1] was limited primarily to single-core security methods, the MRC effort has focused on multi-core methods and diversity. The primary security methods that have emerged are:

1. **Utility Virtual Machines (UVM):** A multi-core operating system organization that separates kernel code into components and protects them through hardware isolation [2,3,4].
2. **Diffusive Scheduling:** A diffusive method for scheduling UVM's on multi-core processors [2].
3. **VT-d:** Methods for leveraging device virtualization technology in UVM's [5].
4. **ExOShim:** A virtualization technology that provides execute-only protection to kernel code [6,7].
5. **Kernel Procedure Linkage Table (KPLT):** A diversification method that redirects references to kernel code in the virtual address of user-processes [8].
6. **Load-time Diversity:** A load-time method, incorporated into an ELF-loader, that disrupts function entry and exit points, with a strategy for using it to diversify hypervisors in addition to kernel and user code [9,10,11,12].



7. **Compile-time Diversity:** A compiler method that disrupts block-level addresses [9,10].
8. **Replicated Diversity:** A method to increase diversity through replication of code [9,10].
9. **Diversified-NFS:** An architectural concept for combining compile-time, load-time, and replicated diversity methods [13].
10. **Asymmetric Multiprocessing:** A research concept that uses asymmetric multiprocessing to improve both security and performance; this early research is not yet complete [14,15].

Other elements of the defense-in-depth approach, begun under the CRASH program have been completed under MRC, and consequently recognize the support, in particular:

11. **Forensics:** A method for discovering zero-day attacks by correlating network traffic with process actions [16,17].
12. **Hiding-in-Hardware:** A Method used to own and control the base-of-trust in hardware and provides hidden hardware monitoring [18,19].

This body of knowledge has been published in 3 Ph.D. theses [2,9,18] and 18 published papers; 2 additional papers are in final stages of preparation/submission [5,12]; two additional Ph.D.'s have been initiated under this funding but are not yet complete.

Other elements of the general approach, funded under CRASH, are briefly referred to here for completeness and context; these include:

- Non-deterministic refresh to deny surveillance, privilege escalation, and persistence.
- Code size and attack surface minimization to reduce vulnerabilities.
- MULTICS-style protection based on 64-bit extended paging tables (EPT).
- Full memory encryption to deny access to code and data in memory and shrink the protection boundary to the chip boundary.
- Network hiding to dynamically change network properties.
- Camouflage to deny system identification.
- Resilience through dynamic process regeneration and remapping to operate through attacks.

*It is not the goal of this report to repeat either the material or the extensive bibliographies provided in the project publications, but rather to provide a cohesive overview of the body of work taken in its entirety.*

The research has been embodied in a clean-slate, multi-core operating system – **Bear** – that operates on Dell workstations (9010/9020), ARM embedded processors (M4/A8/A9), a system-on-a-chip device (Xilinx Zynq), and large-scale blade servers (Dell PowerEdge) [20]. It must be recognized however, that the findings are distributed over this collection of architectures **not** ported to each of them. The reason for this distinction is that, at the time of the research, all of the needed underlying hardware capabilities were not available on any single platform. For example, Intel processors provided virtualization and protection support for guest operating systems (VT-x) and

devices (VT-d); this was not available on ARM processors. Similarly, on-chip encryption/decryption engines and FPGA technology were available on ARM-based devices but not Intel processors with virtualization. Only recently, has there been a confluence of these technologies, with both Intel and Xilinx recently announcing *future* offerings that will combine *all* the needed capabilities into a single processor design – thereby opening the door to an eventual integration of the techniques within a single operating system.

### 3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

**3.1 Assumptions.** At the outset of the project, team members already had extensive experience with a combination of vulnerabilities, exploitation methods, and TTP's that have the DARPA designation "Advanced Persistent Threat" (APT). Consequently, the starting assumption was a threat-model that directly encapsulates the core notions of this designation as illustrated in Figure 1. An APT involves several steps that may include *surveillance* to determine if a vulnerability exists [21], use of an appropriate exploit or other access method [21], privilege escalation [22], removing exploit artifacts, and hiding behavior [23]. Surveillance may involve obtaining a copy of the binary code and using reverse engineering [24,25] or fuzzing [26] to facilitate a broad range of attack vectors including return oriented programming [27]. The implant then *persists* for a time sufficient enough to carry out some malicious effect, obtain useful information, or propagate intrusion to other systems. The ubiquitous use of a small number of operating system types and versions in distributed systems and clouds, has the effect of *amplifying vulnerabilities*: an exploit developed against one version may be used against any host using a similar version. A central goal of this project is to mitigate vulnerability amplification through diversification methods.

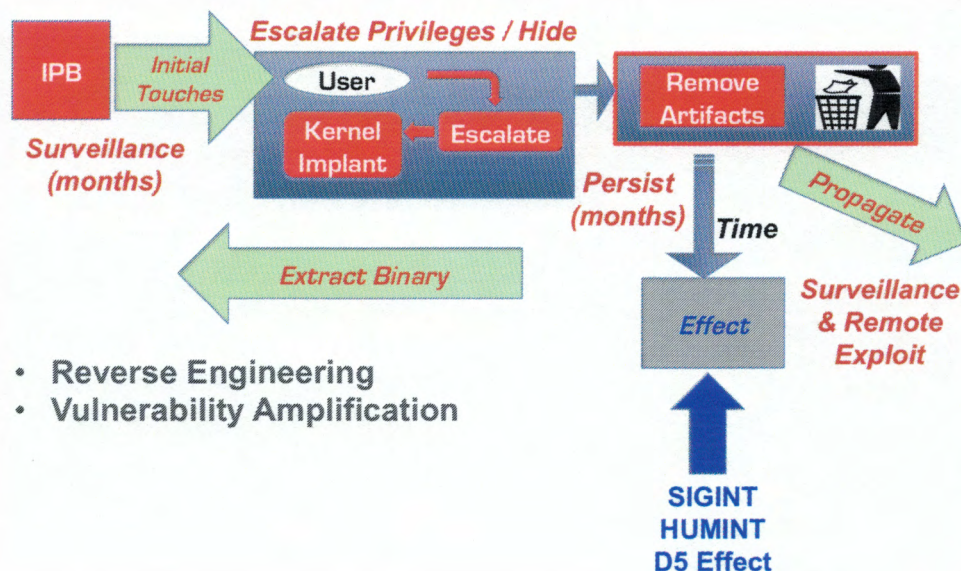
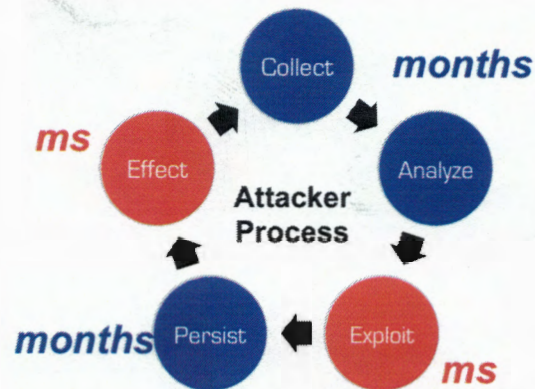


Figure 1: APT Threat Model

A central aspect of this attack process is *timeliness*: the value of information is always qualified by time. For example, a cyber attack assigned to discovery of an opponents air targets would have little utility if not able to provide information within the 24 - 72 hr timeframe covered by air tasking orders (ATO). Figure 2 shows the attackers process from the perspective of timeliness.



**Figure 2: Timeliness in the Attack Process**

Unlike the time to execute an exploit or effect, the time spent in surveillance and persistence may range from minutes to *months or even years* depending upon the intended effect. Moreover, the presence of an intrusion may *never* be detected by network defenses but instead may be recognized indirectly due to either a deviation from expected behavior, the adversary's execution of some D5 effect, or derived from other intelligence sources (SIGINT, HUMINT, etc). Unfortunately, it is precisely the short-timescale areas designated in Figure 2 that are the domain of anomaly and rule-based intrusion detection systems (IDS) and the associated correlation tools. For rule-based detectors, there is no defense against *zero-day attacks* – if an exploit has not been used before, there will be no rule or derivative rule that renders it detectable. Sadly anomaly detectors also fail due to a sad truism: *Not all malicious attacks are anomalous, and not all anomalies are malicious*. In other words, good APT's will hide their behavior and false alarms will obscure their activities.

Current operating system designs have sought to utilize a *static* base of trust and extend trust into software through deliberate layering to combat such threats [28]. Unfortunately, a wide variety of vulnerabilities have appeared that undermine kernel security allowing attackers to implant code, hide, and persist at the highest levels of privilege [29]. The number of vulnerabilities is directly correlated with the size of the code base [30], indicating that there is substantial value in the intellectual process of *reducing the attack surface*; most current operating system designs run into millions of lines of code. Moreover, they compound the opportunity for compromise by granting device drivers unnecessary levels of privilege in order to attain, what in recent years has become, diminishing returns in performance.



## 3.2 Methods and Procedures.

### 3.2.1 Core Ideas. Our core ideas build upon those enumerated in detail in the associated CRASH project [1]. It is not our intent to repeat that material here, but rather to briefly review the ideas in this section for background and context.

Recall that our approach assumes that adversaries will conduct *surveillance*, will be successful in gaining access, will obtain critical system code for reverse engineering, and will *persist undetected* to carry out effects at a later date. To mitigate the risks associated with APT's, we non-deterministically discard the current kernel, user processes, and device drivers. They are replaced by new instances, bootstrapped in the background from read-only gold standards. The cumulative effect of this change in design style is to *increase attacker workload* by continually invalidating surveillance data and denying persistence over time-scales consistent with tactical missions. Unlike other approaches to computer security, no attempt is made to detect intrusions: instead, we focus on continually validating, preserving, and re-establishing the ability of a mission to proceed.

These concepts have been incorporated into a 64-bit version of the *Bear* operating system [20]. The system is composed of a minimalist *micro-kernel* with an associated *hypervisor* that share code extensively to reduce the attack surface. The core functions of scheduling user processes and protecting them from each other are handled by the micro-kernel. All processes and layers are hardened by strictly enforcing MULTICS-style read, write, and execute protections that became available with 64-bit x86 address translation hardware.

All potentially contaminated user processes, device drivers and services are executed with user-level privileges and are strictly isolated from the micro-kernel via a message-passing interface. A notional system task mediates between processes and the kernel to implement the interface. Unlike a conventional rendezvous mechanism in which processes block until synchronization, this asynchronous buffered design provides a single uniform treatment of system calls, inter-process, and inter-processor communication. The interface also supports distributed computing through an MPI-like programming model that maps processes to processors using a user level demon, *rMP*, to provide remote messaging.

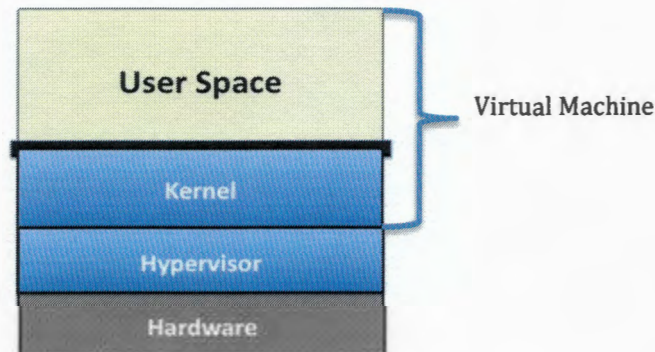
To deny persistence in compromised device drivers and services, the micro-kernel randomly and non-deterministically regenerates them from gold-standard images resident in a *trusted read-only file store*. This store is realized by loading all system code directly into a read-only RAM-disk using an iPXE NIC-assisted boot process. The file system is accessible only from the hypervisor; however, it could alternatively be realized via read-only memory (ROM) or via an out-of-band, write-enabled channel to flash on new hardware. Unlike the MINIX re-incarnation process, regeneration is carried out without regard to the perceived fault or infection status. User processes can also be refreshed through pre-arranged or designated schedules; for example, every few hours, at night, or just prior to a tactical mission.

To deny persistence in the micro-kernel, it is also non-deterministically refreshed from a gold-standard image in the trusted file store, but by the hypervisor. Unlike traditional hypervisors, which are intended to support a general virtual machine execution environment, in CRASH we provided a minimalist hypervisor designed to support *only* the operations required to bootstrap a single micro-kernel and change its network

properties (e.g. IP & MAC address) so as to invalidate an adversary's surveillance data. The current running and bootstrapping instances of the micro-kernel are isolated in hardware through extended page tables, implemented with Intel VT-x extensions. Similarly, in the CRASH project, the network card was isolated through a mapping scheme that attempted to emulate Intel VT-d extensions.

### 3.2.2 Utility Virtual Machines (Method 1)

Utility virtual machines are an alternative multi-core operating system organization. They are best understood by considering the traditional bare-metal (or Type 1) hypervisor that is used in large-scale cloud computing operations, illustrated in Figure 3. The hypervisor controls all the hardware on a system. Conceptually, it presents a virtual machine abstraction that restricts malicious code, executing within one instance of an operating system, from affecting a different instance. On top of the hypervisor sits one or more guest virtual machine(s), which contain an operating system kernel and its associated user processes. The kernel provides networking, scheduling, and many other key functions. The guest's view of hardware is, however, tightly controlled through the Intel Virtualization suite of VT-x (basic virtualization), VT-d (input output memory management unit virtualization), VT-c (network virtualization), and APICv (Interrupt Virtualization). This provides the hardware isolation necessary to protect other guests from a potentially compromised guest, but does *not* protect data inside of that guest.



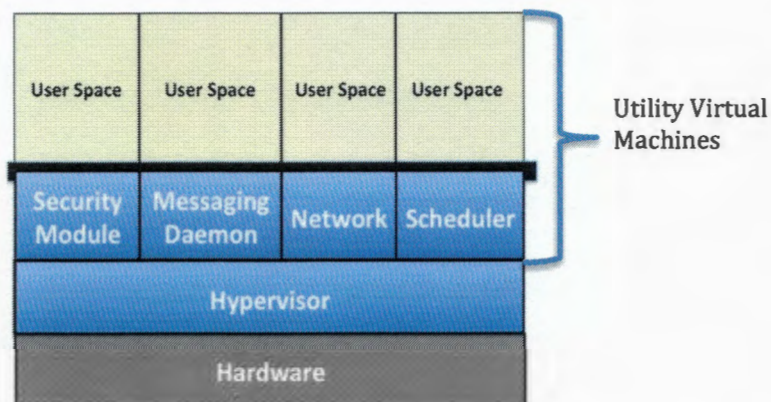
**Figure 3: Bare-Metal Hypervisor**

Unfortunately, hypervisors have continually grown in size and have introduced their own new security challenges: adversaries now actively attempt to detect the presence a hypervisor in order to tailor attacks accordingly. A wide range of hypervisor detection techniques have already appeared against popular systems such as VMWare, VirtualPC, Bochs, Hydra, Xen, and QEMU.

As the number of available cores on a processor continues to expand, it is clearly time to rethink the idea of running a monolithic kernel on a hypervisor. Instead, our Utility Virtual Machine concept separates and isolates specific services as illustrated in Figure 4. Each distinct component of the kernel is protected through *hardware isolation* provided by virtualization. This includes, but is not limited to: user process scheduling, networking, messaging, and other security functions. In a monolithic design, a



compromise of a component, such as the network driver, would give the attacker complete control of the guest and all of its data, which would include many other operating system specific services. Using UVMs, the attacker would only have isolated access to the small subset of data available to the networking UVM, by virtue of hardware protections between virtual machines.



**Figure 4: System based on Utility Virtual Machines**

These ideas have been implemented in a multi-core version of the Bear operating system under MRC program. To eliminate the micro-kernel, and replace it with a collection of UVMs, three key challenges were resolved: isolation of system functions within separate UVMs, communication and synchronization between virtual machines, and the allocation of virtual machines to processing cores to balance load across the cores. Unlike MINIX, which uses the standard rendezvous method to implement system calls and inter-process communication, recall that Bear employs an asynchronous model similar to the message passing interface (MPI) used in parallel and distributed computing. This is central to our design of utility virtual machines since all inter-process communication is transformed into message passing between utility virtual machines. An asynchronous model reduces blocking and allows a higher degree of overlapping across distributed multiprocessors. Obviously, the message-passing interface requires additional compute cycles inside the hypervisor that might be expected slow the guest operation to some degree. However, since guests are no longer full-fledged kernels, their compute requirements and memory footprint is considerably different: What were once solely kernel cycle times are now split with hypervisor cycle times.

To implement UVM's required a complete re-write of the Bear memory system to provide efficient multi-core memory management. This is now achieved using a novel technique known as *recursive paging*, where a single page table entry is reserved to allow fast access to portions of the virtual address space. The technique leverages the hardware features of the memory manager to provide virtual memory mappings on demand. The hypervisor makes use of this method to provide memory sandboxing through Extended Page Tables (EPT) for each virtual machine, whereas the micro-kernel leverages it to enable multiple concurrent processes that operate independently from each other. The UVMs apply the technique to provide the necessary memory for device drivers that they encapsulate.

Symmetric multiprocessing (SMP) is typically enabled through the presence of an Advanced Programmable Interrupt Controller (APIC) inside the silicon of each core on the system. The APIC provides interrupt routing, but more critically supports interrupt routing between cores. The hypervisor hooks onto these interrupts between cores to provide message passing between virtual machines. This hooking is provided through a technique known as APIC Virtualization: any virtual machine that generates any type of interrupt through its APIC, causes the hypervisor to intercept the interrupt. Only when a UVM message is needed does the hypervisor leverage the hook into the inter-core interrupt system. The core where the UVM message is sent, via inter-core interrupt, is determined through the introspection of the virtual machine. However, the cost of using just the inter-core interrupts requires that the hypervisor must also process *all* of the guest interrupts. This task is challenging because each core in the system can generate tens of millions of local interrupts per second through their individual local APIC timers. Therefore, the hypervisor must efficiently process and route every local interrupt and only when needed generate an inter-core interrupt for UVM messaging. Failure to do so can detrimentally impact the runtime performance of a virtual machine executing on the hypervisor.

A messaging system for the UVM framework is only useful if multiple virtual machines can be executed concurrently on top the hypervisor. Moreover, the hypervisor must be able to identify a virtual machine to facilitate where to route the UVM messages. The solution to these issues is to track a variety of information inside the hypervisor. This includes the cores assigned to all virtual machines at any given time and fine grain details of the general purpose register state in each running core. By storing this information, the hypervisor maintains an independent core state for all virtual machines, which allows them to run independently. This information also speeds the process of introspecting virtual machines to acquire UVM message data and to quickly determine where to route a UVM message.

The transfer of messages between virtual machines is achieved by combining inter-core interrupt hooking with the data stored in the hypervisor. The need for this connection is driven by the fact that all virtual machines are sandboxed in memory by their own EPTs, which (intentionally) prevents them from communicating directly. Thus, the hypervisor must operate as the intermediary between these systems. For example, let us assume that a virtual machine running on a core wants to print a character to the screen through the use of an I/O UVM (encapsulating the keyboard and video drivers) running on a different core. First, the virtual machine generates a UVM message send request, which is caught by the hypervisor. The hypervisor then introspects this virtual machine to obtain the message data, which is stored in the UVM messaging system contained in the hypervisor. The hypervisor then initiates an inter-core interrupt to signal the I/O UVM to generate a message receive request. This inter-core interrupt causes the I/O UVM to set up a receive buffer for an incoming message. Then it enters the hypervisor, which performs introspection to determine where the associated buffer is located. The hypervisor then copies the previously stored message into the buffer and resumes execution of the I/O UVM, causing the character to be printed to the screen.

A critical aspect of the UVM concept, cloud computing, and computing in general, is the need for efficient, reliable, and scalable scheduling of system processes. These factors become more critical for UVMs since two levels of scheduling now exists:

one UVM may be running a scheduling algorithm needed to provide fairness to its own user processes while the hypervisor below it may also be running a scheduler to determine which UVM to service.

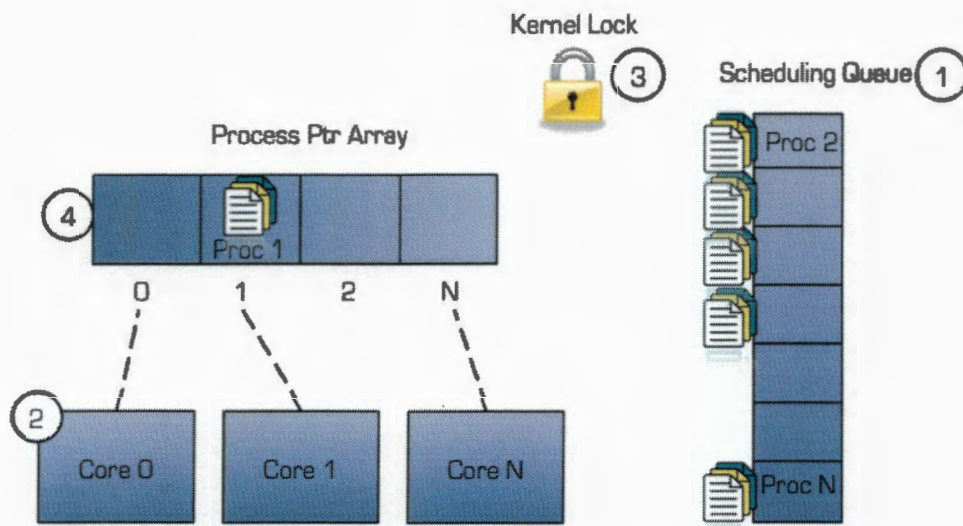
### 3.2.3. Diffusive Scheduling (Method 2)

There exist a wide assortment of scheduling algorithms that can be utilized in operating system design and no one size fits all. For example, real time computing systems must respond to high priority jobs as soon as they occur, because not doing so could result in system failure. For these types of environments, preemptive schedulers are used to give preference to highest priority jobs first and the lowest priority jobs last. In contrast, larger operating systems often use multilevel feedback queues, which partition the ready queue into two or more queues. For each new process that is scheduled the system determines which queue to place the process in. Each queue may have its own unique scheduling algorithm based on the processes it serves. Additionally, this allows an under served process to be rescheduled in a higher priority queue and likewise an over served process to a lower priority queue. However, The main goal of all of these algorithms is to minimize resource starvation, i.e. when a process is denied access to a resource it needs to finish execution.

In the context of UVM's, the critical resource is CPU cycles needed to execute user, kernel, or hypervisor code. The first version of Bear was a uniprocessor system that ran a handful of drivers (executed with user level privileges) and user programs. Fairness was provided to each through a simple round-robin scheduling algorithm. This provides a starvation free solution by offering every process the same time slice to run on the processor core before the next process is scheduled. The enforcement of time slices was provided through the Programmable Interrupt Timer (PIT), which fired at a constant time interval.

As Bear matured, new hardware mechanisms were utilized to replace legacy systems. The most impactful changes to scheduling were the replacement of the PIT with the higher resolution APIC timer and the transition to SMP. The APIC Timer allows scheduling of processes to occur at a faster rate than allowable with the PIT. Additionally, the APIC architecture allows for the scheduling of processes across all of the cores available. This was not possible in the early versions of Bear that utilized the PIT. Nonetheless the round-robin scheduler can still be used with SMP, as illustrated in Figure 5, using a standard method that employs a single kernel lock. All processes that are able to run are stored in ready queue (1). When any of the cores (2) generate a timer interrupt they grab a kernel lock (3) and pull a process from the ready queue (1) in first-in first-out fashion. The process that was previously running on that core is stored in the process pointer array (4); each core has its own entry in the array based on its core number. The previously running process is then put at the end of the ready queue (1). The next process to run is then stored in that core's process pointer array (4) entry. Lastly, the kernel lock (3) is released and the new process executes. Every core uses this scheduling method independently each time a timer interrupt is received.





**Figure 5: Scheduling Software Components**

In addition to architectural changes, such as SMP, added under the MRC project, the user level component of Bear received several new and complex drivers, including a Network File Sharing Daemon (12,850 lines of code) and an e1000 network card driver (939 lines of code) with an associated *LWIP* network stack (33,762 lines of code). These form the backbone for network connectivity and file sharing. From the size of these three components alone, it can be seen that considerable additional resources were required. Ever more challenging operating system concepts in diversity, memory security, and utility virtual machines were explored as described in later sections. Through all of this change, the round-robin scheduler remained in place. Thus, all performance enhancements over this period came from architectural changes that resided below the round-robin code.

Eventually, a new scheduler was sought to better make use of these new realities and improve scheduling performance. Unfortunately, this would not be without its own set of challenges, specifically *process affinity*. This is a uniquely multiprocessor problem based on the principle of cache coherency. In the Intel i7 architecture, the cache is laid out so that each core has its own L1 and L2 cache and all cores share an L3 cache. When a process moves from one core to another, information about it is often shared through the L3 cache; this process is known as snooping. Unfortunately, when data is not propagated fast enough between cores, a cache miss can occur, which introduces a significant time penalty on execution. The processor often has to reach out to main memory to find the needed data, which is a much slower process than when it is available in the cache directly. Another issue involves the introduction of the e1000 network card and its driver into the system: the card itself, by default, generates a hardware interrupt once every 256 nanoseconds (or every 3.9 million cycles of core execution). The APIC timer is typically set to fire once every 34 million cycles. Consequently, the core that receives the network interrupts will have each of its time slices interrupted on average ~8 times. Consequently, any process running on that core will receive less execution time than had it been on another core. This breaks the principle of fairness (i.e. equal time slice) in round robin scheduling.

To address these issues and improve overall system performance a method of scheduling based on *heat diffusion* was incorporated. Previously, much of the supporting research had been performed on large-scale parallel and distributed computing systems. In these studies, one or more compute nodes would quickly become burdened with very large workloads, which then diffused to other nodes via communication. Several beneficial criteria exist for its selection: it uses a simple, fast, scalable algorithm involving only nearest neighbor communication, while global progress and convergence are guaranteed through well-established mathematical analysis. The algorithm has been shown, through simulation, to balance multiple independent load distributions over large-scale distributed architectures, even with huge random load injections. Vector based extensions to the algorithm allow multiple resources (including bandwidth, latency, memory use, and CPU load) to be balanced concurrently.

Similar principles apply to SMP systems: a single compute node can now be considered a single processor core. However, some adjustments must be made to the load calculation to account for traditional measures that cannot be used to determine heat in a localized system. Bandwidth and latency can be eliminated as the cores have near instantaneous communication between each other via an internal crossbar. All cores share the main memory of the system, which removes the need to account for memory usage: no additional memory is available. New measures for load can be attributed to process priority, interrupt routing, and individual core load. Driver processes can be given priority by weighting them at different heat levels than those of a standard user process as they often times perform more complex tasks. In terms of routing interrupts, the core receiving them will by default run hotter than one that is not. Lastly, each process itself carries its own heat that adds load to a core. These three variables can be stored and accumulated to calculate the scalar heat of any core running at any given time. A core can use this heat value to dynamically offload a process to another core with a lower workload.

Two components exist for mapping heat to a processor core and then diffusing work between them. The first is a static component corresponding to initialization and assumptions made for interrupts, process priority, and individual process heat. The second component corresponds to the dynamic load-balancing component that moves processes between cores. These two pieces were incorporated into the round-robin scheduler to provide a diffusive scheduler. The ready queue can continue to store all of the runnable processes that are present on the system by making two minor changes: The first is the addition an identifier in the process structure for each process that maps it to the core it is bound to. This allows for individual processes to be tracked across cores for heat calculations and scheduled by their assigned core; The second modification is the replacement of the *qget()* function, used to retrieve a process from the ready queue, with a *qremove()* function for scheduling the next process. Where *qget()* returns values from the queue in first-in first-out fashion, the *qremove()* function allows the ready queue to be searched by each core via their core ID, which maps to the new identifier in the process structure. The abstract code for this process can be seen in Figure 6.

```

0: static int assigned_core(void* proc, const void* core_id){
1:
2:  /* return first found process assigned to this core */
3:  return ((Proc_t*)proc)->core == (*(uint32_t*)core_id);
4: }
5:
6: Proc_t *ksched_schedule() {
7:
8:  Proc_t *next; /* Next process to run */
9:  uint32_t core = this_cpu(); /* Core requesting next process */
10:
11:  next = (Proc_t*)qremove(readyq, &assigned_core, &core); /* Get next process to run */
12:
13:  -----
14:  -----
15:  /* Other non-diffusion scheduling tasks */
16:  -----
17:  -----
18:
19:  return next;
20: }

```

**Figure 6: Code to Schedule Next Process**

The function *ksched\_schedule()* is called for every timer interrupt to retrieve the next process to run for a specific core. It relies on reading the core's local APIC ID (line 9) to pass to *qremove()* (line 11) along with the global pointer to the ready queue, and the helper function *assigned\_core()*. The sole purpose of the helper function is to return the pointer to the first found process in the ready queue that has been mapped to that core. The process is then removed from the ready queue by the *qremove()* function. Lastly, not seen here, the previously running process is added back to the end of the ready queue.

To initialize the heat map an array of integers is used. Its length corresponds to the number of cores present on the system (8 cores on Dell 9010). All of the cores start with an initial heat value of zero. Cores that handle hardware interrupts can then be assigned heat values of 0, 10, 100, or 1000. These heat values move with the interrupt they are assigned to. Driver processes, like the e1000 driver are assigned heat values of 1 or 10 and move with them as well. All other user space processes are assigned a heat value of 1. Lastly, when a new process is created it is always assigned to the core that created it through the fork system call.

The movement of processes to a new core occurs through the dynamic load balancing code, which is called during a timer interrupt, but before the next process is retrieved through *ksched\_schedule()*. To ease explanation of how this code works, the base case of all processes having a heat of 1 and no interrupt heat assignment is shown in Figure 7. The *balance()* function returns the ID of the core the process will run on the next time it is scheduled. The ID returned by it is stored in the process identifier that was added to the process structure. This is accomplished by assigning the heat value of core zero to a comparator (line 3). Next, the loop (line 5) iterates over the remaining values stored in the heat map. Along the way, if the current comparator's heat is greater than another core's heat, it will then swap the lower heat into the comparator (lines 7-8).



Furthermore, the ID of the core with the lower heat is then stored in the variable *ret* (line 9). Upon completion of the loop the core with the least heat is increased by 1 (line 13). The core the process just ran on has its heat decreased by 1 (line 14).

```

0: int balance(uint32_t core_id){
1:   int i, ret, cmp; /* i to iterate over heat map, ret core id to return, cmp for comparison */
2:
3:   cmp = heat_map[0]; /* start at beginning of heat map for comparison */
4:
5:   for(i = 1, ret = 0; i < smp_num_cpus; i++){
6:     /* if the current map location's load is greater than another's */
7:     if((cmp - DELTA) > heat_map[i]){
8:       cmp = heat_map[i]; /* swap to lower heat map location */
9:       ret = i; /* update ret to reflect this is now the least loaded core */
10:    }
11:  }
12:
13:  heat_map[ret]++; /* increase heat of core process will run on next */
14:  heat_map[core_id]--; /* decrease heat of core the process just ran on */
15:
16:  return ret; /* return the core id the process will run on next */
17: }

```

**Figure 7: Dynamic Load-Balancing Code**

There are a number of ways that this base case can be extended. For example, the process structure can also be passed into the *balance()* function. This allows the routine to check the heat of individual processes for comparison and swapping. So if a driver process with a heat of 10 was being considered for movement, the left half of the *if* statement (line 7) is modified to subtract that processes heat from the comparator (*cmp - DELTA - process\_assigned\_heat*). This also means that a similar change is made to the final addition and subtractions (lines 13 – 14) such that the process heat is accounted for correctly (*heat\_map[ret] += process\_assigned\_heat*, etc). This is just one type of modification that can be made, but other possibilities exist to find the optimal load-balancing solution.

One facet that has not been discussed is the *DELTA* value (line 7) used in the comparator portion of the balancing routine. This value exists due to the process affinity problem and was only discovered through experimentation. The primary purpose is to eliminate cache thrashing across cores in situations when low loads exist. A good explanation of what happens without a delta value occurs when there are 10 processes and 8 cores. In this situation the first 8 processes will be scheduled on one of the 8 cores. The last two processes after each scheduling round will be swapped dynamically to one of the other six. Every time one of these swaps occurs, the next run of that process will result in cache misses and large performance penalties. Experiments with low values for delta appear to indicate that a value of 2 provides a practical situation where processes are *pegged* to processors unless the processor is overwhelmed.

### 3.2.4 VT-d (Method 3)

A longstanding challenge in our research has been the effective use of *peripheral device virtualization*, typified by Intel's VT-d extensions. This technology has been the subject to rapid architectural evolution in recent years, due to increased awareness of the vulnerabilities associated with device drivers. The advances have resulted in poor documentation that is overly complex, contradictory, poorly explained, and incomplete. The current version of Bear uses a full VT-d implementation of its E-1000 network driver and was incorporated at the request of DoD partners and funded independently by them. We are currently check-pointing our experiences in developing the driver for the Intel X86-64 platform and its VT-d support within the Bear hypervisor. The driver was integrated as a user-level daemon, running a full network stack running on top of the Bear microkernel. The implementation explored the concept of non-deterministically refreshing the driver from a gold-standard image to deny persistence using VT-d. The implementation details illuminate the raw capabilities of the hardware, highlights practical challenges, and alerts other developers to the gaps in documentation accessible only through careful exploration of Intel's example code.

### 3.2.5 ExOShim (Method 4)

Information leakage and memory disclosure are major threats to the security in modern computer systems. If an attacker is able to obtain the binary-code of an application, it is possible to reverse engineer the source-code, uncover vulnerabilities, craft exploits, and patch together code-segments to produce code-reuse attacks. These issues are particularly concerning when the application is the operating system kernel itself, because they open the door to privilege escalation and exploitation techniques that provide kernel-level access. ExOShim is a 325-line, lightweight "shim" layer, that uses Intel's commodity virtualization features (extended page tables and protection bits) to prevent memory disclosures by rendering all kernel code *execute-only* i.e. explicitly not readable or writable. This technology, when combined with nondeterministic refresh and load-time diversity explained below, prevents disclosure of kernel code on time-scales that facilitate kernel-level exploit development. Additionally, the shim employs self-protection and hiding techniques to guarantee its operation even if the attacker gains full kernel level access. The proof-of-concept prototype incorporated into Bear was evaluated using metrics that quantify its code size and complexity, associated run-time performance costs, and its effectiveness in thwarting information leakage. Unlike other approaches, ExOShim is the first to provide complete execute-only protection for kernel code and has a runtime-performance overhead of only 0.86%. The concepts are general and could also be applied to other operating systems. The ExOShim functionality has been explored in two variants: one where it is slipped underneath a running kernel, the other involves direct integration within the hypervisor.

### 3.2.6 KPLT: Diversity through a Kernel Procedure Linkage Table (Method 5)

It is standard practice in modern operating systems for the kernel to be mapped into the virtual address space of every user process for efficiency; after all, invariably every process needs access to the kernel's functionality at some point in its execution. The

kernel is never considered “shared” memory in the conventional sense, but upon closer inspection, the kernel does show similar attributes. In particular, the kernel is commonly mapped at the *same location* for every user process. Consequently, its location is predictable, giving rise to *privilege escalation* opportunities. During the MRC program, we explored the idea of treating the kernel as a shared object and forcing a level of indirection, through virtual addressing, in order to access it. This is achieved by interjecting a Kernel Procedure Linkage Table (KPLT) into each process that maps the same (shared) kernel functions into different virtual addresses in each process. The framework and mechanisms that use the table protect it from discovery, while allowing the KPLT to increase diversity by changing the addresses of kernel functions on a per-process basis.

### 3.2.7 Load-time Diversity (Method 6)

Our primary goal in developing other diversity techniques were to ensure that the following properties were achieved:

1. All function entry points are disrupted.
2. All function exit points are disrupted.
3. All basic blocks (*if*, *while*, *switch*, etc.) within functions are disrupted.

The first of these properties eliminates entry into malicious code by patching a predictable address; the second eliminates the opportunity to return to normal operation from malicious code; the final property ensures that every instance of a function has a unique layout i.e. all jump offsets within all basic blocks are diversified. In combination, when applied to an operating system binary code, these properties ensure that *no two instances of a running operating system share the same exploitable address* – thereby eliminating vulnerability amplification in clouds.

The standard Executable and Linkable Format (ELF) format, used in the program compilation and linking process, segregates an executable program into distinct sections that designate TEXT (code), DATA (initialized variables), RODATA (read-only data), and BSS (uninitialized variables). The resulting ELF file also contains headers that describe how these sections should be stored in memory. Typically, for example in Linux, functions are loaded sequentially into sections and sections are loaded back-to-back into memory. There is no re-ordering of functions or sections and as a result, the location of code in memory is deterministic, predictable, and can be reverse-engineered. Instead, we force the compiler to build a separate section for each function using the standard *-ffunction-sections* compiler option. This allows our diversifying ELF loader to re-order function layout, placing each function in a random page at load time. Moreover, using *relocations* (Rel/Rela sections) generated by the compiler, the loader is able to update inter-section dependencies between functions and data at load-time.

Load-time diversity involves dispersing functions randomly across the entire virtual address space at load-time to achieve properties 1 and 2 above, using the underlying paging system. During the project, a variety of algorithms were explored to achieve this dispersion; the final version -- affectionately referred to by the members of the group as “Uberdiversity” -- uses almost the entire x86-64 virtual address space. To implement this technique, the separate ELF sections are randomly distributed over the

virtual address space at load-time by a specialized diversifying ELF-loader. This loader is responsible for patching up function calls and function returns to take account of the dispersion achieved during loading using relocations. The loader uses the Intel x86-64 hardware virtual memory abstraction to inject the maximum amount of entropy into a loaded program instance. A particularly novel aspect of the approach is that it is incorporated into a standard stage 2 bootloader; this allows kernel and user code to be interspersed with each other with appropriate protections. In addition, for the first time, this has made it possible to diversify the hypervisor itself, within its own separate address space. A detailed theoretical study of diversity has been conducted to explore the limits of the approach both from the viewpoint of the degree of entropy introduced (i.e. the likelihood that an instruction falls at a predictable location) and the number of unique memory layouts that a particular application admits to.

### 3.2.8 Compile-time Diversity (Method 7)

Recall that the load-time diversification technique described in section 3.2.7 satisfies the first two desired diversification properties, concerning the disruption of function entry and exit points, but not the last, associated with disrupting basic code blocks: The loader modifies address references *between* sections, but not *within* sections. Furthermore, keep in mind that there is a loss of 12 potential bits of entropy at load-time due to constraints on the paging system (i.e. caused by page size and alignment constraints) – *more than three orders of magnitude!* To resolve these issues a compile time source-to-source transformation is used that adds *vacuous code padding* to basic blocks, achieving the final 3rd property of disrupting basic blocks while gaining back this lost entropy.

To inject entropy into every logical block in a program, a random number of bytes, between 0 and  $2^s - 1$ , are injected into the beginning of every logical block in the program, using a uniform random distribution. The inserted bytes themselves are random numbers. Jump instructions are inserted before the random byte stream to ensure it is not actually executed at run-time; this minimizes the performance effect of the transformation, typically the entire block is in cache. The insertion is achieved through a source-to-source transformation on the original C source code using a Clang compiler plug-in.

Figure 8 shows the padding transformation in action. The left-most function in Figure 8 is the original source code; the two right-most functions demonstrate two possible results from the source-to-source transformation: the first using a two-byte, followed by a four-byte random sequence; the second using a one-byte followed by a two-byte sequence. The consequence of this transformation is that if an attacker attempts to use pre-existing code present in the binary, based on static analysis, the execution will incorrectly jump to a random prior location, typically causing a crash. On some rare occasions where a crash is not triggered, an unexpected non-deterministic action will be performed. Alternatively, traps could be placed in the random sequence to allow detection.

The parameter  $s$  is configurable and can be provided with separate values for the blocks that represent function opening and the beginning of other logical blocks. For function opening, the maximum useful value for  $s$  is 12; buying back the entropy lost to paging and alignment constraints. If the beginning of each function is located at the beginning of a page, as is the case with the ELF loader, this entropy is additive with the entropy injected by the ELF loader. Any larger value of  $s$  would overlap with the relocation provided by the functional loader, resulting in no appreciable gain in entropy.

The entropy injected into the jump offsets of logical blocks is, assuming a uniform distribution on  $[0, 2^s - 1]$ , simply equivalent to the value of  $s$ . The limitation is the overhead willing to be accepted in file and memory size increases; for our systems, we typically use a value of 8 (giving 256 possible variants for each jump offset).

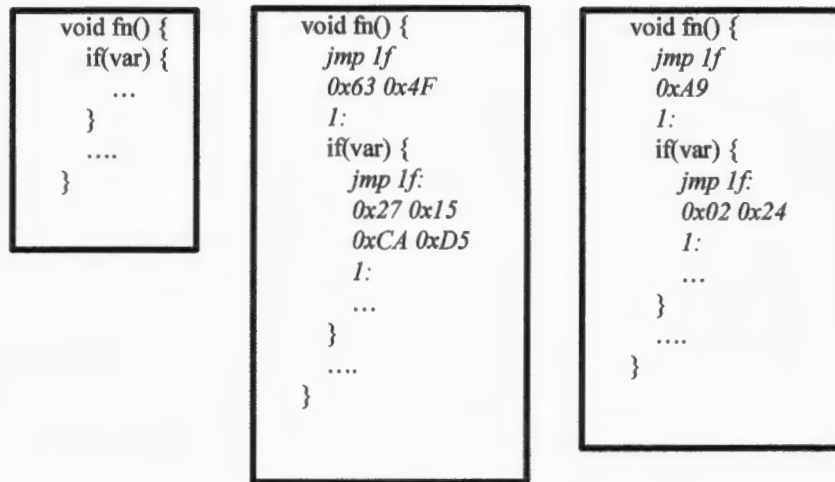


Figure 8: Original Function with two vacuous padded variants.

### 3.2.9. Replication Diversity (Method 8)

To augment the runtime and compile time transformations described thus far, we have developed a runtime transformation based on *function replication*. In this transformation, functions (or more specifically, relocatable code units) are cloned at runtime. Any calls into a unit are redirected at random into one of the clones.

The effect of this transformation is to increase the likelihood that if an attacker were to fortuitously discover a usable address, there is no guarantee that the address would be *consistently* usable at run-time. While it might work acceptably in a return-oriented programming system, if the address was used to resume normal execution it would not be guaranteed to operate correctly. Furthermore, an active intrusion detection system could be created to utilize the clones – for example, if an unexpected clone was ever executed, it would be clear that at some point the program execution had been derailed.

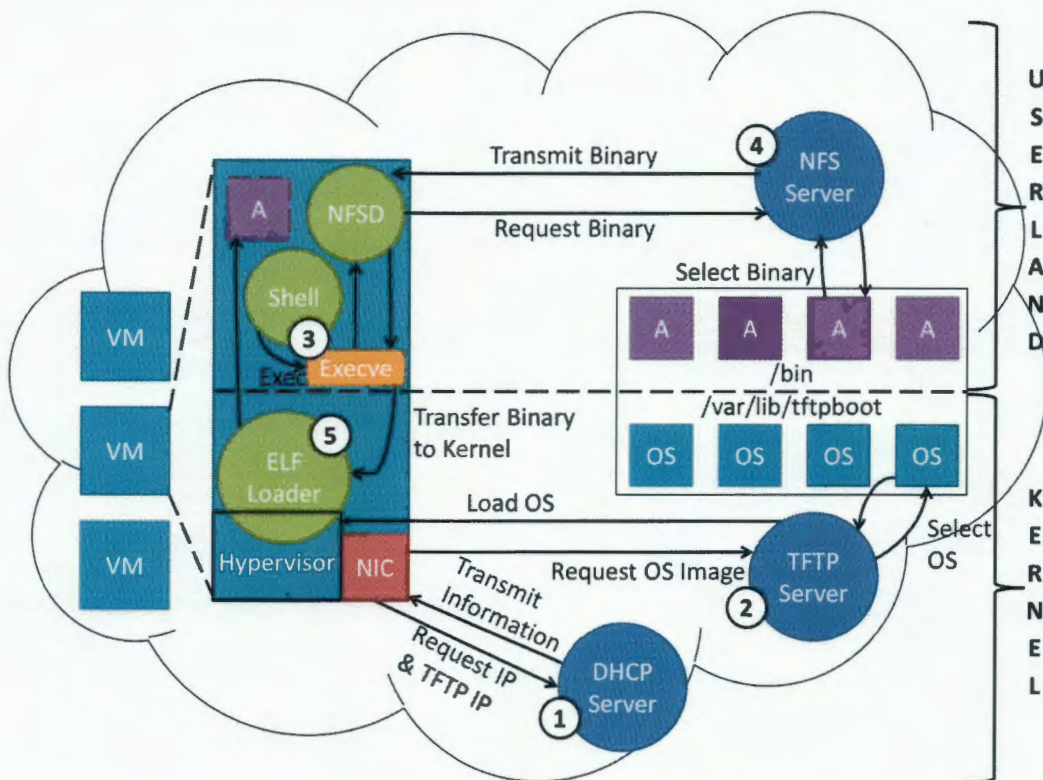
This transformation, for modest levels of replication (e.g. 3) increases memory usage, but has negligible impact on the performance. The clone to use is picked at random before execution is initiated, and then never changed. Relocations, jumps, and control flow instructions are unmodified; only endpoint addresses are reconfigured. At every refresh, a different one of the replicas is used.

### 3.2.10 Diversified-NFS (Method 9)

Figure 9 illustrates Diversified-NFS – an architectural organization for combining compile-time, load-time, and replication diversity explained in previous sections. It operates in combination with the Preboot eXecution Environment (PXE-boot) and Dynamic Host Configuration Protocol (DHCP). The overall concept places DHCP, TFTP



(used by PXE-boot), and NFS servers on a separate out-of-band network or V-LAN within the cloud that is used only for code protection. Each *program*, corresponding to an application, operating system, or hypervisor, is compiled at an out-of-band server to form a *variant repository* within the distributed file system. For the sake of simplicity, we currently place all application binaries in /bin; operating system and hypervisor binaries are placed in /var/lib/tftp. A background task at the out-of-band server periodically recompiles each program and replaces the existing version within the repository to avoid the delay associated with on-demand compilation. The random nature of the compile-time diversity transformation ensures that each time a program is recompiled a unique binary variant is available for loading as indicated by the multiple instances (in time) of an application (A) and operating system (OS).



**Figure 9: D-NFS System Overview**

When a physical host bootstraps within the cloud, its NIC card obtains an IP address from the DHCP server together with the address of the TFTP server (1). The NIC card then downloads the current OS variant comprising a RAM-disk (containing the compile-time diversified variants of the hypervisor and microkernel) from the TFTP server (2). It then bootstraps the hypervisor using the diversifying ELF-loader (load-time diversity), choosing a random function replica (replication diversity). This ensures that each *instance of an operating system variant* has a unique image in memory. Services and device drivers, loaded during bootstrapping, are also diversified with the modified ELF loader.

The bootstrapped operating system incorporates an NFS client, designated in Figure 9 as NFSD. Eventually, the operating system opens a shell and begins execution of commands (3) using an *execve* system call. This call will load the current variant of the application binary code over the network through NFS (4) into the operating system kernel. The diversifying ELF-loader is then used to initiate execution of the application (5). This ensures that the each *instance of an application* in memory is random and unique.

This process mitigates *vulnerability amplification* and substantially *increases attacker workload*: in order to craft a reusable exploit, the attacker must reverse-engineer each program variant and determine its unique memory footprint, on each host in the cloud, *as a function of time*. The more frequently a program is reloaded, the more difficult the attackers task. The intent is to force the refresh frequency inside (shorter than) the time to reverse-engineer a code and develop an exploit. This removes the opportunity to use an exploit without the need to detect intrusions. The approach yields the full capacity of entropy for even small code bases, allows compile-time code-size overhead and entropy to be adjusted based on threat-level, and has negligible run-time overhead; all of the overhead is paid up-front at load-time. We have demonstrated the system by running all components within the same cloud at Rackspace, and by running the NFS, DHCP, and TFTP servers remotely at Dartmouth, with the rest of the system at Rackspace. By replacing PXE with iPXE, it is possible to encrypt and sign binaries downloaded from the repository.

### 3.2.11 Asymmetric Multiprocessing (Method 10)

We have recently been exploring a new operating system design that completely decouples the kernel from user processes. This is achieved by running the kernel and user processes on separate processor cores instead of at different privilege levels on a single core. Rather than the traditional approach of using interrupts to implement system calls, we instead utilize the hardware facilities for inter-core communication developed as part of our UVM research. Surprisingly, on modern processors, this now offers the opportunity for increased performance, while providing a hard separation between user processes and the kernel. One of the central advantages of this approach is that it then allows user processes and device drivers to be elevated to operate in at a higher privilege level. This offers device drivers the performance normally associated with monolithic systems, but provides the security generally associated with micro-kernels.

## 4.0 RESULTS AND DISCUSSION

### 4.1 Core Results: Utility Virtual Machines

Two test suites were used to benchmark the memory and processor performance of our preliminary UVM implementation. Performance is measured in processor cycles using the time stamp counter, which counts the number of cycles executed since a core starts. The time for a test is calculated by dividing the total cycles needed for completion of the test by the speed of the processor.

The memory benchmark used was developed by Chuck Lever and David Boreham at the University of Michigan and measures the performance of `malloc()` in a multithreaded system. This benchmark supports the use of either threads or processes to test physical memory performance. POSIX threads are not supported in the current Bear micro-kernel implementation; in common with Linux, lightweight processes provide a more uniform programming abstraction. For the testing reported here, 100 processes are created that execute a loop running for 100 million iterations; each iteration executes one `malloc()` and `realloc()` on a 1024 byte block size.

Processor performance is measured using the addition, subtraction, and multiplication modules from the popular AIM9 synthetic benchmark suite. The AIM9 suite specifically tests processor performance by executing instructions that stress test the internal processor logic. For the testing reported here, a single process runs 100 iterations of each of the AIM9 addition, multiplication, and division benchmarks.

Two test systems were used for the benchmarks: a Dell OptiPlex 9010 with 4GB RAM and an 8-core 3.4 GHz Intel i7 processor, and a MacBook Pro with 8GB RAM and a 3.2GHz processor. The Dell system ran the Bear Micro-Kernel, Bear Micro-Kernel on its Hypervisor, Fedora with 3.17.4-301 Linux Kernel, and Fedora 3.17.4-301 Linux Kernel on Xen 4.4 Hypervisor. The MacBook Pro ran VMware Fusion, a type 2 hypervisor [16] with an Ubuntu with 2.6.32-38-generic Linux Kernel guest that has 4Gb of ram and 4 processor cores provided to it from the MacBook Pro.

Table 1 below provides the average processor cycles and time from twenty runs of the above-mentioned tests. The table itself is broken into the Cycles it took to complete the memory benchmark, AIM9 benchmark, and total for both. The Table also provides the time in seconds it took to complete the respective benchmarks and total time for both. There are several interesting observations from this table. Using the memory and recursive paging system discussed in this paper, the micro-kernel is 34.4% faster than the Fedora Kernel. Furthermore, The micro-kernel on the custom hypervisor is 27.7% faster than Fedora on Xen, and 25.9% faster than Ubuntu on VMware. Some of this performance gain should be attributed to the fact that a micro-kernel is a much lighter weight operating system than a full Linux kernel and thus can create processes at a faster rate. However, the purpose of the benchmark in creating 100 processes with a large number of `malloc()` and `realloc()` iterations is to focus the performance measurements more on the operating systems use of `malloc()`, `realloc()` and virtual memory for user space as a whole. This provides a level of certainty in the performance gains provided by a recursive paging system.



	Memory Cycles	AIM9 Cycles	Total Cycles	Memory Time (s)	AIM9 Time (s)	Total Test Time (s)
Bear Micro-Kernel	2.95738 E+11	1.44554 E+11	4.40292 E+11	86.981	42.515	129.497
Bear Micro Kernel & Hypervisor	2.99564 E+11	1.54416 E+11	4.5398 E+11	88.107	45.416	133.523
Fedora Kernel	4.79616 E+11	1.43682 E+11	6.23298 E+11	141.063	42.259	183.322
Fedora Kernel - Xen Hypervisor	4.07463 E+11	1.92471 E+11	5.99935 E+11	119.842	56.609	176.451
Ubuntu Guest - VMware Fusion	4.60572 E+11	9.38998 E+10	5.54472 E+11	143.928	29.343	173.272
Bear System with UVM	3.02883E+11	1.4533E+11	4.48212E+11	89.083	42.744	131.827

**Table 1: Memory and Processor Benchmarks**

Processor performance based on the AIM9 benchmarks had three interesting comparison points. While it was expected that the Bear micro-kernel would outperform the larger Fedora Kernel as it did in memory, this is not the case. In fact, both systems scored roughly the same in cycles and time, with the Fedora kernel edging out the micro-kernel by ~.251 of a second to complete the AIM9 benchmark. This is attributed to the minimalistic nature of the micro-kernel and the superior scheduling offered by Fedora, which for both means the AIM9 test is running at all times on a singular core.

It is important to notice the impact that a newer processor has on the AIM9 benchmark. The type 2 VMware hypervisor running Ubuntu is running on a slower processor and with 4 less cores, but that processor was released ~15 months after the processor shipped with the Dell. The difference a year can make is staggering, as the Ubuntu guest finishes the AIM9 benchmarks almost a full 13 seconds faster than any configurations running on the Dell.

The presence of a hypervisor slows performance of the AIM9 benchmark on all of the systems. The micro-kernel has the smallest impact, which is due to configuring the hypervisor to operate the guest as close to real time as possible. Larger hypervisors such as Xen and VMware are designed to manage multiple guests, some configurations that are suitable to a micro-kernel are not appropriate for them. This can be seen in the larger performance impact when comparing the Fedora kernel to the Fedora kernel on the Xen hypervisor.

Lastly, the performance of a stripped down prototype UVM messaging system that contains just the keyboard & VGA drivers was also evaluated. As expected it performs about equal to the micro-kernel with hypervisor. Being about ~1 second slower in memory performance and ~3 seconds faster in AIM9 performance. Noticeably, the AIM9 performance is expected to slow in future development cycles. As more

functionality will be added to the system through the addition of a greater number of UVMs.

The benchmarks demonstrate that well-known optimization techniques, used in today's state-of-the-art systems can make a significant difference in performance. Although using only basic scheduling concepts and little in the way of optimization, the micro-kernel design performs surprisingly well when compared with mature systems that have undergone hundreds of man-years in development and optimization. This can be directly attributed to extensive use of Intel hardware mechanisms using their recommended implementation methods to build an SMP enabled hypervisor and micro-kernel.

## 4.2 Diffusive Scheduling

To eliminate as many external factors that could impact performance, experimentation on diffusive scheduling was completed on the kernel only version of the system. This removes the slow-down generated by the presence of the hypervisor and the virtual APIC settings. The memory benchmark is well suited for the evaluation of the diffusion scheduler as a single process spawns 100 additional processes. This results in one core having a high initial load that it then transfers to the other cores.

The initial run of the diffusion scheduler used the code seen in Figure 7 without the *DELTA* variable. The AIM9 test suite, which runs as a single process, illustrates the problem of process affinity as cache thrashing occurs and results in high overheads. Once, the issue was noticed, a *DELTA* of one and two are used in all further testing. Additional configurations of the scheduler include drivers with heat values of 10, all drivers pegged to a core, and hardware interrupts of heat 10, 100, or 1000. The results of the varying methods and the round-robin scheduler performance are seen in Table 2.

When reviewing Table 2, it is important to examine the individual tests first when evaluating the new scheduler, as the memory test benefits from improvements to multi-process execution. Whereas the AIM9 test suite sees performance gains when single process execution is sped up through an enhancement. Adding the times it takes to complete both tests together provides an overall measure of performance, but may miss potential impacts to either form of execution.

For example, on the surface the diffusive scheduler without the *DELTA* variable overall performs 15.68% worse than the round-robin scheduler. This is solely because of a 42.78% performance penalty taken during single process execution of the AIM9 suite due to cache thrashing. In fact, multi-process execution during memory testing is improved by 1.32%, which almost certainly is impacted by cache thrashing to some degree. Thus, all testing is performed with a *DELTA* present. As noticeable performance gains were only shown using a *DELTA* of two, the following discussions will only be in regards to that setting.

The diffusive scheduler performs equivalently to the round-robin scheduler once process affinity has been accounted for. Further exploration of increasing process affinity was explored by pegging drivers to a single core. This alone did not result in any performance gains. In an attempt to isolate drivers further from user tasks, their heat value was increased from 1 to 10. As this resulted in a .52% speedup for memory, a .97%

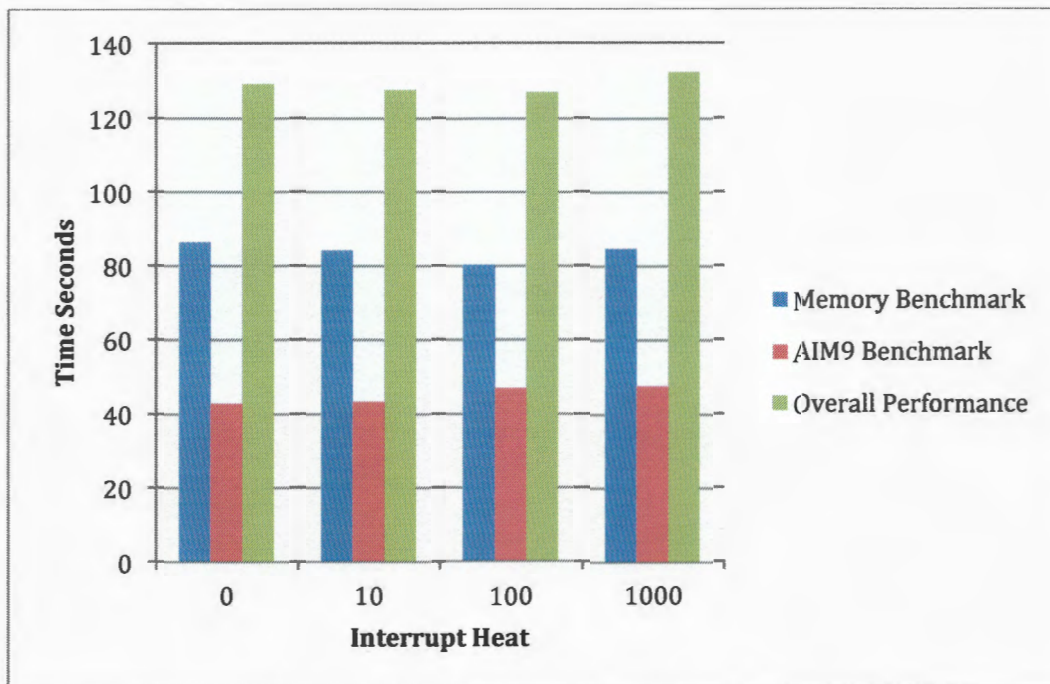
speedup for AIM9, and an overall speedup of .70%. Unfortunately, these results provide less than a 1% margin for scheduling improvement.

Scheduler Configurations	Cycles Memory	Cycles AIM9 Add/Mul/Div	Cycles Total	Memory Time (s)	AIM9 Time (s)	Total Time (s)
Round-Robin Scheduler	2.9574E+11	1.4455E+11	4.4029E+11	86.98	42.54	129.50
Diffusion – All Processes 1	2.9185E+11	2.2333E+11	5.1519E+11	85.84	65.69	151.53
Diffusion – All Processes 1, Delta 1	2.8910E+11	1.7981E+11	4.6891E+11	85.03	52.88	137.91
Diffusion – All Processes 1, Delta 2	2.9378E+11	1.4500E+11	4.3878E+11	86.41	42.65	129.05
Diffusion – All Processes 1, Delta 1, Peg Drivers	2.9111E+11	1.7725E+11	4.6836E+11	85.62	52.13	137.75
Diffusion – All Processes 1, Delta 2, Peg Drivers	2.9316E+11	1.4593E+11	4.3909E+11	86.22	42.92	129.14
Diffusion – User Processes 1, Delta 1, Peg Drivers 10	2.9391E+11	2.0721E+11	5.0112E+11	86.44	60.94	147.39
Diffusion – User Processes 1, Delta 2, Peg Drivers 10	2.9420E+11	1.4323E+11	4.3743E+11	86.53	42.13	128.66
Diffusion - User Processes 1, Delta 2, Interrupts 10, Peg Drivers 1	2.8634E+11	1.4769E+11	4.3402E+11	84.22	43.44	127.65
Diffusion – User Processes 1, Delta 2, Interrupts 100, Peg Drivers 1	2.7283E+11	1.5956E+11	4.3239E+11	80.24	46.93	127.17
Diffusion – User Processes 1, Delta 2, Interrupts 1000, Peg Drivers 1	2.8835E+11	1.62252E+11	4.5060E+11	84.81	47.72	132.53

**Table 2: Scheduler Performance Characterization**



The last variable that impacts normal core execution is hardware interrupts. In this measurement the core receiving hardware interrupts from the I/O APIC will be assigned a heat of 10, 100, or 1,000. Drivers will continued to be pegged to cores to improve their individual process affinity as experiments showed a marginal benefit in doing so. The graphed results of increasing heat can be seen in Figure 10.



**Figure 10: Diffusion Performance as Interrupt Heat Rises**

Looking at the graph it can be seen that from 0 to 100, memory and overall performance improve, while AIM9 performance decreases as interrupt heat rises. Going from 100 to 1000 heat causes a decrease in all three categories. However, the setting of this variable at 100 has an effect of improving multi-process performance by 8.06%, but decreasing single process performance by 9.81%, which amounts to an overall performance increase of 1.82%. These results are noteworthy as they clearly demonstrate that the heat diffusion algorithm has a marked scheduling improvement in environments with heavy workloads.

With all of the changes to the scheduler, it might be expected that there was a large addition to the number of lines of code, which would increase the attack surface. However, this was not the case, by re-purposing pieces of the original round-robin scheduler, the need for additional code was low: To add the minimal amount of support for diffusion, 29 lines of code were needed for the configuration presented in Figure 7; To go to the full interrupt and driver pegging setup requires only an additional 23 lines of code, which brings the total to 52 lines. The brunt of this work was in developing an alternative conceptual framework and adapting it to operate on multiple cores.

## 5.0 CONCLUSIONS

The concepts and technologies developed under this project extend and harden the defense-in-depth strategy developed in the associated CRASH project to the domain of symmetric multiprocessing and cloud computing. Utility Virtual Machines serve to separate concerns, use hardware mechanisms to harden boundaries between operating systems functions, and reduce the overall attack surface. Diffusive scheduling improves performance of UVM implementations in high-load situations and sets up the opportunity to transparently schedule resilient applications among multi-processors. Several varieties of diversification allow us to throttle vulnerability amplification and increase the workload of reverse engineering and exploit development; when combined with dynamic refresh these ideas introduce a time-dependent element into the randomization process that substantially increases attacker workload.

This project affirms and solidifies our general conclusions from the CRASH program: Military systems have gained tremendously from the cost and flexibility benefits afforded by widespread adoption of commercial off the shelf (COTS) technology -- to the point where it is now difficult to imagine how we might operate, with similar levels of assurance and efficiency, using non-COTS methods. However, in times of tension, critical mission capabilities *must* continue to operate, even if major components of "the network" are unavailable and the systems upon which we rely are repeatedly compromised by error, fault, or malicious action. It therefore behooves us to apply Occam's razor to pare back the layers of complexity that have been thrust upon us by commercial vendors, in light of the controlled environment in which DoD operates, to selectively improve *resilience* and *increase attacker workload*.

Our approach is to use COTS subsystems, accepting their imperfections, but augmenting them with ideas from the fault-tolerance, distributed computing, and encryption communities. The body of research explored how we might pursue this goal using three basic non-deterministic precepts:

- Don't trust what you have – *continually validate, replicate and regenerate,*
- Don't advertise what you do – *continually hide and camouflage,* and
- Don't be predictable – instead be *diverse, mobile and non-deterministic.*

The Bear system uses overlapping regenerative techniques, combined at every layer of the system, from the user to the hardware. These methods deny surveillance and throttle vulnerability amplification by continually invalidating surveillance data, randomizing systems across memory, hiding in the network, and using camouflage. Persistence is denied by non-deterministically replacing, refreshing, replicating, diversifying, and/or relocating components so as to continually re-establish trust. The methods can be incorporated individually, as independent modes, or collectively and continuously for critical missions.



## 6.0 REFERENCES

- [1] S. Taylor, "Attacking Time: Mitigating Advanced Persistent Threats (APT's)", Apr 3, 2015, Final report: DARPA CRASH Program, Air Force Research Laboratory, Rome, NY.
- [2] R. Denz, Securing Operating Systems Through Utility Virtual Machines, Ph.D. Thesis, Thayer School of Engineering at Dartmouth College, June 2016.
- [3] R. Denz, and S. Taylor, "A Survey on Securing the Virtual Cloud", Journal of Cloud Computing: Advances, Systems, and Applications, Volume 2, Issue 1 on 6 November 2013.
- [4] R. Denz and S. Taylor, "Securing the Cloud through Utility Virtual Machines", In the Proceedings of IMCIC, Orlando, FL, March, 2016.
- [5] S. Kuhn and S. Taylor, "VT-d: Revealing Complexity and Pitfalls", In preparation.
- [6] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code", 11th International Conference on Cyber Warfare and Security (ICWS'16), pp 56–64, Boston University, Boston, MA, March 2016.
- [7] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code.", International Journal of Information and Computer Security, April 2016.
- [8] S. Brookes, M. Osterloh, R. Denz, and S. Taylor, "The KPLT: The Kernel as a Shared Object", MILCOM 2015, pp 981-986, Oct 2015.
- [9] M. Kanter, "Enhancing Non-determinism in Operating Systems", Ph.D. Thesis, Thayer School of Engineering at Dartmouth, October 2013.
- [10] M. Kanter, and S. Taylor, "Diversity in Cloud Systems through Runtime and Compile-Time Relocation", In proceedings of IEEE-HST 2013.
- [11] M. Kanter, and S. Taylor, "Attack Mitigation through Diversity", In proceedings of MILCOM 2013, pp 1410-1415, Nov 2013.
- [12] S. Brookes, M. Osterloh, R. Denz, and S. Taylor, "Überdiversity", In preparation.
- [13] M. Osterloh, R. Denz, and S. Taylor, "Diversified-NFS", ICCSM 2014, Oct, 2014.
- [14] S. Brookes and S. Taylor, "Rethinking Operating System Design: Asymmetric Multiprocessing for Security and Performance", New Security Paradigms Workshop 2016, Sept 26-29 2016. Accepted for Publication.

- [15] S. Brookes and S. Taylor, "Containing a Confused Deputy on x86: A Survey of Privilege Escalation Mitigation Techniques", *International Journal of Advanced Computer Science and Applications (IJACSA)*, April 2016.
- [16] S. Kuhn and S. Taylor, "Locating Zero-day Exploits with Course-Grained Forensics", Expanded version of conference paper by same name. *Journal of Information Warfare*, Vol 14, Issue 4, Oct 2015.
- [17] S. Kuhn and S. Taylor, "Locating Zero-day Exploits with Course-Grained Forensics", 14th European Conference on Cyber Warfare and Security ECCWS-2015, 2-3 July 2015, pp 159-168.
- [18] J. Dahlstrom, "Hiding in Hardware", Ph.D. Thesis, Thayer School of Engineering at Dartmouth College, December 2015.
- [19] J. Dahlstrom and S. Taylor, "Hardware-Based Code Monitors on Hybrid, Processor-FPGA System-on-Chip Architectures, *MILCOM* 2015, pp 968-973, Oct 2015.
- [20] C. Nichols, M. Kanter, and S. Taylor, "Bear – A Resilient Kernel for Tactical Missions", In proceedings of *MILCOM* 2013, pp 1416-1421, Nov 2013. (Co-funded with CRASH)
- [21] D. Kennedy, J. O’Gorman, D. Kearns, and M Aharoni, "Metasploit: The Penetration Testers Guide", No Starch Press, 2011.
- [22] L. Davi, A Dmitrienko, AR Sadeghi, M Winandy, "Privilege Escalation Attacks on Android", *Information Security*, Springer 2011.
- [23] Greg Hoglund and Jamie Butler, "Rootkits", Addison-Wesley Professional Press, 2005.
- [24] C. Eagle, "The IDA Pro Book", No Starch Press, 2011.
- [25] Eldad Eilam, "Reversing", Wiley, 2005
- [26] J.E. Forrester and B.P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing", *4th USENIX Windows Systems Symposium*, Seattle, August 2000. Appears (in German translation) as "Empirische Studie zur Stabilität von NT-Anwendungen", *iX*, September 2000.
- [27] S. Checkoway, A.J. Halderman, A. J. Feldman, E. W. Felten, B. Kantor, and H. Shacham (2009); "Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage", in *Proceedings of the USENIX/ACCURATE/LAVoSS Electronic Voting Technology Workshop*, August 2009.

- [28] W. A. Arbaugh, D. J. Farber, and J. M. Smith. "A secure and reliable bootstrap architecture." In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (SP '97). IEEE Computer Society, Washington, DC, USA, 65-. 1997.
- [29] B. Blunden. *The Rootkit Arsenal: Escape and Evation in the Dark Corners of the System*. USA: Jones and Bartlett Publishers, Inc. 2009.
- [30] Pandey and Tiwari, "Reliability Issues in Open Source Software." *International Journal of Computer Applications*, vol. 34 issue 1, pp. 34-38. 2011.

## 7.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

APIC	Advanced Programmable Interrupt Controller
APT	Advanced Persistent Threat – cyber implant that persists and hides
ARM	Advanced RISC Machines – a type of computer processor
ATO	Air Tasking Order
CPU	Central Processing Unit
COTS	Commercial of the shelf
D5 Effects	Deceive, Deny, Disrupt, Degrade, Destroy
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
ELF	Executable and Linkable Format
EPT	Extended Page Tables
FPGA	Field-Programmable Gate Array
HUMINT	Human Intelligence
IDS	Intrusion Detection System
IOMMU	input/output memory management unit
IP	Internet Protocol
KLPT	Kernel Procedure Linkage Table
MAC Address	Media Access Control Address – identifies a network interface
MINIX	mini-Unix -- a micro-kernel based operating system
MMU	Memory Management Unit
MPI	Message Passing Interface -- software API
MULTICS	Multiplexed Information and Computing Service – an operating system
NFS	Network File System
PIT	Programmable Interrupt Controller
POSIX	Portable Operating System Interface
PXE	Preboot eXecution Environment
RAM	Random Access Memory
ROM	Read-only memory
ROP	Return Oriented Programming – a form of cyber attack
rMP	resilient Message Passing software system
SIGINT	Signals Intelligence
SMP	Symmetric Multiprocessing
TFTP	Trivial File Transfer Protocol
TTP	Tools, Techniques, and Procedures -- operational aspects
UVM	Utility Virtual Machine
VLAN	Virtual Local Area Network
VT-c/d/x	Intel virtualization technologies